Reasoning about code

CSE 331
University of Washington

Michael Ernst
Reasoning about code

Determine what facts are true during execution

\[ x > 0 \]

for all nodes \( n: n.next.previous == n \)

array \( a \) is sorted

\[ x + y == z \]

if \( x \neq \text{null} \), then \( x.a > x.b \)

Applications:

Ensure code is correct (via reasoning or testing)
Find defects
Reproduce failures
Understand why code is incorrect
Verify a representation invariant

Does this code maintain the rep invariant?

class NameList {

    // representation invariant: 0 ≤ index < names.length
    int index;
    String[] names;

    ...

    void addName(String name) {
        index++;
        if (index < names.length) {
            names[index] = name;
        }
    }
}
What must the caller do?

Incompletely documented:

```java
// @param name a full name, last name first, like "Doe, John"
// @returns a two-element array of the first name and last name
String[] parseName(String name) {
    int commapos = name.indexOf(",");
    String lastName = name.substring(0, commapos);
    String firstName = name.substring(commapos + 2);
    return new String[] { firstName, lastName }; 
}
```

- What input produces [“John”, “Doe”]?
- What input produces [“ohn”, “Doe”]? [“ John”, “Doe”]?
- How can you improve the precondition?
Web server using SQL database

String userInput = …;
String query = "SELECT messages FROM users "+ "WHERE name='" + userInput + "'";
statement.executeUpdate(query);  // execute DB query

Is it possible to retrieve information for all users?
query = "SELECT messages FROM users WHERE name='a' or '1'='1";

User inputs: a' or '1'='1
query = "SELECT messages FROM users WHERE name='a' or '1'='1"

http://xkcd.com/327/

Automatic Creation of SQL Injection and Cross-Site Scripting Attacks

Adam Kiezun
MIT
akiezun@csail.mit.edu

Philip J. Guo
Stanford University
pg@cs.stanford.edu

Karthick Jayaraman
Syracuse University
kjayaram@syr.edu

Michael D. Ernst
University of Washington
memet@cs.washington.edu

Abstract
We present a technique for finding security vulnerabilities in Web applications. SQL injection (SQLI) and cross-site scripting (XSS) attacks are widespread forms of attack in which the attacker crafts the input to the application to reflect his own data. Previous approaches to identifying SQLI and XSS vulnerabilities and preventing exploits include defensive coding, static analysis, dynamic monitoring, and test generation. Each of these approaches has its own merits, but also offers opportunities for improvement. Defensive coding [6] is error-prone and requires rewriting existing software to achieve a desired level of security. Static analysis can be ineffective due to its inability to analyze large libraries of code. Dynamic analysis is too slow for widespread use. Finally, test generation is difficult if not impossible given the large number of possible inputs that might result in secure or insecure application behavior.

DID YOU REALLY NAME YOUR SON Robert’); DROP TABLE Students;-- ?

OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.

AND I HOPE YOU’VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.
Ways to get your design right

The hard way

Start hacking

When something doesn't work, hack some more

How do you know it doesn't work?

Need to reproduce the errors your users experience

Apply caffeine liberally

The easier way

Plan first (specs, system decomposition, tests, ...)

Less apparent progress upfront

Faster completion times

Better delivered product

Less frustration
Ways to verify your code

Goal: correct code
The hard way: hacking
  Make up some inputs
  If it doesn't crash, ship it
  When it fails in the field, attempt to debug
An easier way: systematic testing
  Reason about possible behaviors and desired outcomes
  Construct simple tests that exercise all behaviors
Another way that can be easy: reasoning
  Prove that the system does what you want
    Rep invariants are preserved
    Implementation satisfies specification
Proof can be formal or informal (we will be informal)
Complementary to testing
Forward reasoning

You know what is true before running the code
   What is true after running the code?
Given a precondition, what is the postcondition?
Example:
   // precondition: x is even
   x = x + 3;
y = 2x;
x = 5;
   // postcondition: ??
Applications:
   Rep invariant holds before running the code
      Does it still hold after running the code?
   Does a method satisfy its spec?
      If precondition holds, does postcondition hold?
Backward reasoning

You know what you want to be true after running the code. What must be true beforehand in order to ensure that? Given a postcondition, what must the precondition be?

Example:
// precondition: ??
\[
\begin{align*}
x &= x + 3; \\
y &= 2x; \\
x &= 5; \\
\end{align*}
\]
// postcondition: \( y > x \)

What was your reasoning?

Application:
(Re-)establish rep invariant at method exit: what requires? Reproduce a failure: what must the input have been? Exploit a defect
Forward vs. backward reasoning

Forward reasoning is more intuitive

  Simulates the code (‘abstract interpretation’)
  Introduces facts that may be irrelevant to the goal
  Set of current facts may get large

Backward reasoning is sometimes more helpful

  Helps you understand what should happen
  Given a specific goal, indicates how to achieve it
  Given an error, gives a test case that exposes it
Does the postcondition hold?

Use forward reasoning

```java
int x = ...;
int z = ...;
assert x >= 0;

z = 0;  // x ≥ 0

if (x != 0) {
    z = x;  // x > 0 & z = 0
} else {
    z = z + 1;  // x = 0 & z = 1
}

assert z > 0;
```

// (x > 0 & z = x) OR (x = 0 & z = 1)
What input led to assertion failure?

Most common application of backward reasoning

```java
if (x != 0) {
    // x < 0  OR  (x = 0 & z ≤ -1)
    // (x ≠ 0 & x ≤ 0)  OR  (x = 0 & z ≤ -1)
    z = x;
    // x ≤ 0
    // z ≤ 0
}
else {
    // z ≤ -1
    z = z + 1;
    // z ≤ 0
}
assert z > 0;
```
Another example of backward reasoning

// precondition: ??

if (x < 5) {
    x = x*x;
} else {
    x = x+1;
}

// postcondition: x ≥ 9

(x ≤ -3) OR (x ≥ 3 & x < 5) OR (x ≥ 8)
(x < 5 & x*x ≥ 9) OR (x ≥ 5 & x+1 ≥ 9)

x*x ≥ 9
x ≥ 9
x+1 ≥ 9
x ≥ 9

Called the “weakest precondition” or “wp”
If statements review

Forward reasoning

\{P\}

if B

\{P \land B\}
S1
\{Q1\}

else

\{P \land \neg B\}
S2
\{Q2\}
\{Q1 \lor Q2\}

Backward reasoning

\{ (B \land wp(S1, Q)) \lor (\neg B \land wp(S2, Q)) \}

if B

\{wp(S1, Q)\}
S1
\{Q\}

else

\{wp(S2, Q)\}
S2
\{Q\}
\{Q\}
Forward reasoning with a loop

```c
assert x >= 0;

i = x;  // x >= 0 & i = x
z = 0;  // x >= 0 & i = x & z = 0

while (i != 0) {
    z = z + 1;  // LOOP-BEGIN i != 0 & ((x >= 0 & i = x & z = 0) OR LOOP-END)
    i = i - 1;  // LOOP-END
}

assert x == z;  // x >= 0 & i = 0 & z = x
```

Infinite number of paths through this code
How do you know that the overall conclusion is correct?
**Induction** on the length of the computation
Reasoning about loops

A loop represents an unknown number of paths
  Case analysis is problematic
  Recursion presents the same problem as loops
Cannot enumerate all paths
  This is what makes testing and reasoning hard
Things to prove about a loop:
  1. It computes the **correct value**
  2. It **terminates** (no infinite loop)
Reasoning about loops: values and termination

// assert x ≥ 0 & y = 0
while (x != y) {
    y = y + 1;
}
// assert x = y

Does “x=y” hold after this loop?
Does this loop terminate?
1) Pre-assertion guarantees that x ≥ y
2) Every time through loop
   x ≥ y holds before at the test
   If the body is entered, x > y -- this is LOOP-BEGIN
   y is incremented by 1
   x is unchanged
   Therefore, y is closer to x (but x ≥ y still holds) – this is LOOP-END
3) Since there are only a finite number of integers between x and y, y will eventually equal x
4) Execution exits the loop as soon as x = y (but x ≥ y still holds)
Understanding loops by induction

We just made an inductive argument
  Inducting over the *number of iterations*

Computation induction
  Show that conjecture holds if zero iterations
  Show that it holds after \( n+1 \) iterations
  (assuming that it holds after \( n \) iterations)

Two things to prove

1. Some property is preserved (known as “*partial correctness*”), *if* the code terminates
   Loop invariant is preserved by each iteration, if the iteration completes

2. The loop completes (known as “*termination*”)
   The “decrementing function” is reduced by each iteration and cannot be reduced forever
Example: Factorial

```plaintext
{ arg≥0 & n=arg } // n is a temporary variable
r=1;
while (n≠0) {
  r=r*n;
  n=n-1;
}
{ r=arg! }  
```

```
arg≥0 ∧ n=arg ∧ r=1
r=arg!/n! ∧ arg≥n>0
r=arg!/(n-1)! ∧ arg≥n>0
r=arg!/n! ∧ arg≥n≥0
```

```
(n = 0 ∧ arg≥0 ∧ n=arg ∧ r=1) OR (n = 0 ∧ r=arg!/n! ∧ arg≥n≥0)
(n = 0 ∧ arg=0 ∧ r=1) OR (n = 0 ∧ arg≥0 ∧ r=arg!)
```

“Loop invariant”. Where did this come from?
Loop invariant

1. When reverse engineering: *guess* it
2. When designing: choose it *before* writing code

To design loops or recursion:
• Decompose large problems into smaller ones
  – “Divide and conquer” or “Wishful thinking” design methodology
• “I don’t know how to compute $n!$, but I could compute it if you told me $(n-1)!$.”
  – What about 0!?
Loop design methodology

1. Decompose problem into
   – Assumption that most of the problem is solved
   – A small increment of remaining work
2. Write an invariant that expresses the milestone of each iteration
3. Write a loop body to perform the increment while maintaining the invariant
4. Write the loop test so false-implies-postcondition
5. Write initialization code to establish invariant
Loop design example

Set \textbf{max} to hold the largest value in array \textit{items}

\[
\text{max} = \text{amax}(\text{items}[0..\text{len}])
\]

1. Decomposition: Given \( \text{amax}(\text{items}[0..\text{len}-1]) \), can determine \( \text{amax}(\text{items}[0..\text{len}]) \)

\[
\text{amax}(\text{items}[0..\text{len}]) = \max(\text{amax}(\text{items}[0..\text{len}-1]), \text{items}[\text{len}])
\]

2. Invariant: \textbf{max} holds largest value in range \textit{items}[0..k-1]
Loop design example

Set \texttt{max} to hold the largest value in array \texttt{items}

3. Write a loop body to perform the increment and maintain the invariant

\begin{verbatim}
// inv: max holds largest value in items[0..k-1]
while (...) {
    // inv holds
    if (items[k] > max) {
        max = items[k]; // breaks inv temporarily
    } else {
        // nothing to do
    }
    // max holds largest value in items[0..k]
    k = k+1; // invariant holds again
}
\end{verbatim}
Set `max` to hold the largest value in array `items`.

4. Write the loop test so false-implies-postcondition.

```java
// inv: max holds largest value in items[0..k-1]
while (k != items.length) {
    // inv holds
    if (items[k] > max) {
        max = items[k]; // breaks inv temporarily
    } else {
        // nothing to do
    }
    // max holds largest value in items[0..k]
    k = k+1; // invariant holds again
}
```
Loop design example

Set \texttt{max} to hold the largest value in array \texttt{items}

5. Write initialization code to establish invariant

\begin{verbatim}
    k = 1;
    max = items[0];
    // inv: max holds largest value in items[0..k-1]
    while (k != items.length) {
        ...
    }
\end{verbatim}
Loop design edge case

Our initialization code has a precondition: `items.size > 0`

```java
// items.length > 0
k = 1;
max = items[0];
// inv: max holds largest value in items[0..k-1]
while (k != items.size) {
    ...
}
```

Such a (specified!) precondition may be appropriate
Else need a different postcondition (“if size is 0, ...”) and a conditional that checks for the empty case

Or the `Integer.MIN_VALUE` “trick” and logical reasoning

Neat: Precise preconditions should expose all this to you!
Example: Quotient and remainder

Compute quotient and remainder for num/denom

\[
\begin{align*}
q &:= 0; \\
r &:= num;
\end{align*}
\]

while (denom <= r) {
    \[
    r := r - denom;
    q := 1 + q;
    \]
}\]

// num = q × denom + r & r < denom
Example: Greatest common divisor

\{ x_1 > 0 \land x_2 > 0 \}

y_1 := x_1;
y_2 := x_2;
while (y_1 \neq y_2) do
    if y_1 > y_2
        then y_1 := y_1 - y_2
    else y_2 := y_2 - y_1
{ y_1 = \text{gcd}(x_1, x_2) }

Recall: if y_1, y_2 are both positive integers, then:
• If y_1 > y_2 then \text{gcd}(y_1, y_2) = \text{gcd}(y_1 - y_2, y_2)
• If y_2 > y_1 then \text{gcd}(y_1, y_2) = \text{gcd}(y_1, y_2 - y_1)
• If y_1 - y_2 then \text{gcd}(y_1, y_2) = y_1 = y_2
Goal: Demonstrate that rep invariant is satisfied

• Exhaustive testing
  – Create every possible object of the type
  – Check rep invariant for each object
  – Problem: impractical

• Limited testing
  – Choose representative objects of the type
  – Check rep invariant for each object
  – Problem: did you choose well?

• Reasoning
  – Prove that all objects of the type satisfy the rep invariant
  – Sometimes easier than testing, sometimes harder
  – Every good programmer uses it as appropriate
All possible objects (and values) of a type

• Make a new object
  – constructors
  – producers
• Modify an existing object
  – mutators
  – observers, producers (why?)
• Limited number of operations, but infinitely many objects
  – Maybe infinitely many values as well
Examples of making objects

Infinitely many possibilities
We cannot perform a proof that considers each possibility case-by-case
Solution: induction

Induction: technique for proving *infinitely* many facts using *finitely* many proof steps

For constructors ("basis step")
- Prove the property holds on exit

For all other methods ("inductive step")
- Prove that:
  - if the property holds on entry, then it holds on exit

If the basis and inductive steps are true:
- There is no way to make an object for which the property does not hold
- Therefore, the property holds for all objects